Noname manuscript No.
(will be inserted by the editor)

Top-k Queries on Temporal Data

Feifei Li · Ke Yi · Wangchao Le

Received: date / Accepted: date

Abstract The database community has devoted extensive amount of efforts to indexing and querying temporal data in the past decades. However, insufficient amount of attention has been paid to temporal ranking queries. More precisely, given any time instance t, the query asks for the top-k objects at time t with respect to some score attribute. Some generic indexing structures based on R-trees do support ranking queries on temporal data, but as they are not tailored for such queries, the performance is far from satisfactory. We present the SEB-tree, a simple indexing scheme that supports temporal ranking queries much more efficiently. The SEB-tree answers a top-k query for any time instance t in the *optimal* number of I/Os in expectation, namely, $O(\log_B \frac{N}{B} + \frac{k}{B})$ I/Os, where N is the size of the data set and B is the disk block size. The index has near-linear size (for constant and reasonable $k_{\max}$ values, where $k_{\max}$ is the maximum value for the possible values of the query parameter k), can be constructed in near-linear time, and also supports insertions and deletions without affecting its query performance guarantee. Most of all, the SEB-tree is especially appealing in practice due to its simplicity as it uses the B-tree as the only building block. Extensive experiments on a number of large data sets, show that the SEB-tree is more than an order of magnitude faster than the R-tree based indexes for temporal ranking queries.

Feifei Li, Wangchao Le
Computer Science Department, Florida State University, USA
Tel.: +1-850-645-2356, Fax: +1-850-644-0058
E-mail: lifeifei@cs.fsu.edu, le@cs.fsu.edu

Ke Yi
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China
Tel.: +852-2358-8770, Fax: +852-2358-1477
E-mail: yike@cse.ust.hk

1 Introduction

Temporal data refers to objects that change over time. Examples include time series (e.g., stock traces, sensor readings, etc.), trajectories for spatial objects transactional and temporal databases. Due to their numerous applications, efficient managing, indexing, and querying temporal data have been extensively studied. Those attributes of objects in a database that change over time are often referred to as the *temporal attributes*, for instance the temperature attribute in a sensor readings database. In general, a temporal attribute A of an object can be an arbitrary function $f : \mathbb{R} \to \mathbb{R}$, but often time f is represented as a piecewise linear function (i.e., a sequence of straight line segments) for efficient storage, transmission, indexing, and querying [12, 25]. There is a vast literature on how to approximate an arbitrary function f by a piecewise linear function g. Please see [5,12,27,32] and the references therein. Generally speaking, more segments in g lead to better approximation quality, but also increase the size of the representation. Adaptive methods, in which more segments are allocated to regions of high volatility while less segments for smoother regions, are better than non-adaptive methods which use a fixed segmentation interval. Segmentation is beyond the scope of this paper, and we assume that the data has already been converted to a piecewise linear format by *any* segmentation method. We do not even require all the piecewise linear functions have the same number of segments. Thus it is possible that the data is gathered from a variety of sources after different preprocessing modules.

Problem definition. In this paper, we study a particularly useful query type on a collection of objects with a temporal attribute: top-k queries. We define our prob-

lem more precisely as follows. Suppose we have a family of n piecewise linear functions, $\mathcal{F} = \{f_1, \ldots, f_n\} : \mathbb{R} \rightarrow \mathbb{R}$, where f_i *is the temporal attribute of object* o_i *and consists of* m_i *segment*s. We also refer to these functions as *score functions*. Given any t and k, a *top-*$k(t)$ query retrieves the k functions that have the highest function values $f_i(t)$. Ties are broken arbitrarily. The goal is to store these functions in an index structure such that a top-$k(t)$ query can be answered efficiently for any t and $k \leq k_{\max}$ where $k_{\max}$ is the maximum possible k that is supported. We would like to emphasize that the piece-wise linear line segments from different functions are not necessarily aligned along the time axis (i.e., line segments from different functions could start and end at different time instances, see the example in Figure 1), and functions may also have different number of piece-wise linear line segments. We also use *objects* when referring to these piece-wise linear functions.

An example of such ranking queries is illustrated in Figure 1 where Q_1 and Q_2 ask for top-2 objects at time instance t_1 and t_2, respectively. The answer to Q_1 is (o_2, o_1) and it is (o_1, o_3) for Q_2. Note that if we simply attach the object id of o_i to each segment of f_i, the problem is equivalent to the following more general problem. Given a set of $N = \sum_{i=1}^{n} m_i$ segments (n functions and the ith function has m_i segments) in the xy-plane, for a given query coordinate t, return the top-k segments having the highest intersection points with the vertical line $x = t$. We denote this set of segments, the set of all segments from all functions, as S.

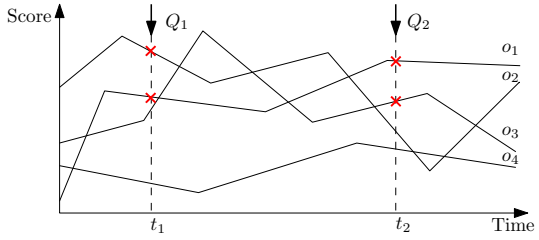


Fig. 1 A top-$k(t)$ query example.

The top-$k(t)$ query has numerous practical applications on temporal data, as illustrated next.

Example 1 A common query on archival stock price data is to retrieve the top-k stocks with the highest prices at a particular time in the past. As storing the stock prices at every second consumes a large amount of storage, a common way to store archival stock price data is to approximate them by segmentation. Thus the above query is exactly a top-$k(t)$ query on piecewise linear functions to be investigated in this paper.

Example 2 Another application is to report the top-k sensors with the highest temperature readings at a certain time instance. Since wireless transmission of data is the biggest source of battery drain, a common way of saving transmission is to approximate the readings by a piecewise linear function and the sensors only send these functions back to central station, as opposed to sending readings continuously at, say every second.

Example 3 Each piecewise linear function could also represent the sale-volume of an product over time. Then a top-$k(t)$ query is especially useful for finding out the k-most popular products at any time instance from a large collection of objects over a long time period.

Note that top-$k(t)$ queries should not be confused with similarity search in time series or trajectories [12, 14, 37]. In the latter case, one is interested in retrieving the k most similar functions f_i to a query function, where the similarity is measured over the entire time domain of the function using for example the L_2 metric.

Numerous efforts have been spent on indexing and querying temporal data, most notably on similarity search aggregate queries, nearest neighbors, range queries, temporal pattern queries, and interval skyline queries. However, we feel that the small amount of attention that top-$k(t)$ queries has received is disproportional to their importance. Indeed, a top-$k(t)$ query is a special one-dimensional k-nearest neighbor query where the the query point is always at infinity, so any temporal k-nearest neighbor index can be used to answer a top-$k(t)$ query as well. But as these indexes are not tailored for top-$k(t)$ queries, the query performance is far from satisfactory (please refer to our detailed discussion on this issue in Section 2). We believe that the importance of ranking queries justifies the need of a specifically designed index that optimizes for such queries.

Our contributions. This paper presents a comprehensive study for top-$k(t)$ queries. We first discuss some special cases and baseline solutions. This includes how to use the asymptotically optimal multiversion B-tree (MVB-tree) for the special case when the score functions are piecewise constant as mentioned above. For piecewise linear functions we briefly discuss how the temporal k-nearest neighbor indexes, which are all based on R-trees, support top-$k(t)$ queries, but these indexes do not have any query performance guarantee. We also discuss how to extend the MVB-tree solution to the piecewise linear function case, but with a quadratic blowup in the index size and the construction cost.

Then, we present the SEB-tree (the *sampled envelope B-tree*), the new index structure for top-$k(t)$ queries. Compared with the baseline solutions, the SEB-tree is superior (or at least on par) in the following aspects.

Query performance: Theoretically, it answers a top-$k(t)$ query in $O(\log_B \frac{N}{B} + \frac{k}{B})$ I/Os in expectation, which

matches the query performance of the MVB-tree for the general piecewise linear functions. The R-tree approaches have no query guarantee at all. Our index is randomized, and the expectation on query cost is w.r.t. the randomization *within* the structure, not in the data or the query. So for R-tree based indexes there are some bad inputs that *always* lead to large query costs, but such bad inputs do not exist for our index.

Size and construction: The SEB-tree takes $O(\frac{N}{B}\alpha(\frac{N}{B})(\sqrt{\log\frac{N}{B}}+\log\frac{k_{\max}}{B}))$ [1] disk blocks in expectation, and requires expected $O(N\alpha(\frac{N}{B})\log\frac{N}{B}(\sqrt{\log\frac{N}{B}}+\log\frac{k_{\max}}{B}))$ time for construction, where $\alpha(\cdot)$ is the inverse Ackermann function, an extremely slow-growing function and can be essentially treated as a small constant for all imaginable data sizes. Thus, the SEB-tree does require more space than the R-tree, but only by a small logarithmic factor, and its construction cost is much less than a R-tree (as seen in our experiments). Though the MVB-tree only takes linear space and $\frac{N}{B}\log\frac{N}{B}$ construction cost, but the the MVB-tree based solution for top-$k(t)$ queries introduces a quadratic blowup to its size and construction cost, which makes it at least an order of magnitude worse than the SEB-tree.

Updates: The SEB-tree supports updates of historical records. It is a sampling based structure, and as a result, for 99.5% of the updates (based on our experimental results), we just need to perform a few insertions or deletions in the collection of B-trees. Thus the concurrency control, logging, and recovery mechanisms of B-trees can still be enforced easily and efficiently. For the rest 0.5% updates, we may need to lock and update a larger portion of the B-tree, but these happen rarely. In contrast, the MVB-tree does not allow any changes to the historical records and updating the R-tree with concurrency control and locking is highly inefficient (due to the re-insertion).

Furthermore, the SEB-tree is in fact just a collection of B-trees and has a simple query algorithm (but with a nontrivial analysis). Thus it can be easily integrated into a commercial DBMS. In contrast, there is so far no commercial implementation of the multiversion B-tree yet, and the support for R-trees is also limited in commercial DBMSs.

We survey the baseline solutions and other related works in Section 2. The SEB-tree is discussed in Section 3, followed by a useful space-query trade-off and the support of ranking with multiple attributes per object and time instance for the SEB-tree in Section 4. An extensive experimental report is presented in Section 5, and the paper is concluded in Section 6.

[1] In this paper we let $\log(x) = \max\{\log_2(x), 0\}$ for all $x > 0$.

2 Baseline Solutions and Related Work

We give some baseline solutions using existing indexing structures and survey the related work in this section.

2.1 Special case, naive and baseline solutions

We first review some baseline solutions to the top-$k(t)$ queries by leveraging on existing techniques. Note that these techniques were not designed for answering top-$k(t)$ queries, nevertheless, we show how to extend them to solve the top-$k(t)$ queries proposed in this work.

Staircase score functions and the multiversion B-tree. If the temporal attribute of each object is a piecewise constant (i.e., staircase) function rather than piecewise linear, the problem can be efficiently solved using the *multiversion B-tree* [8], which answers a range query at any time instance in the optimal $O(\log_B\frac{N}{B}+\frac{k}{B})$ I/Os, where B is the block size. Unfortunately, it is not known how to extend the multiversion B-tree to the case where the functions are piecewise linear. Furthermore, the multiversion B-tree is an inherently static structure that does not support insertions and deletions of historical records: Updates are allowed only on the current version of the B-tree.

We first show, how the multiversion B-tree (MVB-tree) [8] can be used to solve the special case where all the score functions are piecewise constant (i.e., staircase). An example is illustrated in Figure 2. In this case, if we sweep a vertical line from left to right, the n objects become one-dimensional objects on the sweeping line whose locations could be updated for a total of N times. We observe that this is exactly what the MVB-tree is designed for. The MVB-tree stores in a compact way all the N versions of the B-tree, one after each update. It supports queries on any version of the B-tree as efficiently as if each version is stored individually. A top-$k(t)$ query retrieves the k highest objects at time t, namely, from the version of the B-tree at that time. Thus, the query can be answered in the optimal $O(\log_B\frac{N}{B}+\frac{k}{B})$ I/Os using the MVB-tree.

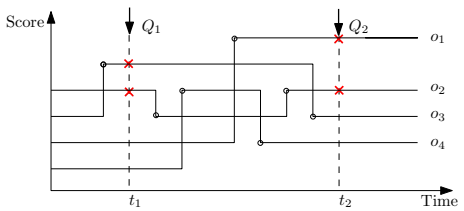


Fig. 2 Ranking queries on objects with staircase functions.

It is well known that for a large number of typical functions a piecewise linear function has a much higher approximation accuracy than a staircase function, given the same description size. Unfortunately, the MVB-tree

does not generalize to our general setting with piecewise linear functions, the fundamental reason being that the MVB-tree essentially keeps all versions of the B-tree. For staircase functions, there are only $\Theta(N)$ different orderings, but this number becomes $\Theta(N^2)$ for piecewise linear functions, which would be prohibitive for the MVB-tree to store. Furthermore, the MVB-tree does not support insertions/deletions of objects or changes in their score functions.

Naive solutions. There are two naive solutions to the top-$k(t)$ queries. The first solution is to based on the temporal data using the original time instances, instead of representing them with piece-wise linear functions. In this case, one can pre-process all time instances and sort all objects at each time instance, then store the sorted orders for every time instance. Then, one can index all time instances using an B-tree (time instance as the indexing attribute and the sorted list of object ids at a particular time instance as the indexing record). The obvious benefit of this approach is the low query cost and ease of implementation. However, this approach does not scale since the space consumption is linear to the number of time instances for the union of all objects, which is significantly higher than the space consumption of a solution based on the piece-wise linear representation of these objects. This also eliminates all the benefits of the piece-wise linear segmentation as discussed in Section 1. The expensive storage cost also introduces overheads to the query performance, since the B-tree has to index much more (easily by a few orders of magnitude) points than the solution based on the piece-wise linear functions. Furthermore, this solution does not support dynamic updates efficiently.

The second solution applies the above idea on the piece-wise linear functions to reduce the space consumption. It works as follows. Given the piece-wise linear representation for all temporal data objects, each object could be viewed as a piece-wise linear function defined by multiple linear line segments (as shown in Figure 1. In a pre-processing step, we can find all the intersection points among all linear line segments for all objects. Every two intersection points define an interval. Next, we sort all objects in each interval and index these sorted lists by an B tree where the indexing attribute is the x-values of the intersection points that define these intervals. Unfortunately, this approach too suffers from the high space consumption, even though it does reduce the storage cost comparing to the first solution above. In the worst case, given a total of N linear line segments in the union of all objects, there could be $\Theta(N^2)$ intersection points defined as above. Hence, this approach requires *quadratic* space with respect to the input data set. It also introduces higher query cost since the B tree has to index $\Theta(N^2)$ keys.

This solution is essentially the generalization of the MVB-tree approach on the staircase functions to the general piecewise linear functions. Specifically, for any input, the MVB-tree solution needs to find all the possible intersection points among all segments of all objects, which naturally leads to $O(N^2)$ smaller segments to be indexed, on a typical input. Since the MVB-tree takes linear space to the data to be indexed, the MVB-tree based solution will take $O(\frac{N^2}{B})$ space and $O(\frac{N^2}{B}\log\frac{N}{B})$ construction cost. Its query cost is just $O(\log\frac{N}{B})$, which is similar to the query cost of the SEB-tree.

Piecewise linear score functions and R-tree based indexes. For piecewise linear score functions, almost all existing techniques rely on the R-tree, *the* generic multidimensional spatial index. The observation is that if we treat each f_i as the trajectory of a one-dimensional object over time, then the top-$k(t)$ query becomes k-nearest neighbor (kNN) query at time instance t using a query point that is high enough. The state-of-art techniques on indexing trajectories [5, 16, 19] all share the following framework. We first break up each trajectory (score function) into a number of pieces, then an R-tree is built on top of these pieces. After we have such an R-tree, the generic kNN algorithm can be applied to answer a query at any given time instance t. Essentially, one uses the same branch-and-bound kNN search principle in R-trees. The R-tree nodes, starting with the root node, and the trajectories in the leaves are explored in the increasing order of their "minimum snapshot distance" (denoted as the MinSTDist, see Figure 3) using a priority queue. The search terminates when there are k objects whose MinSTDist are larger than any objects not seen. This R-tree based algorithm is the baseline solution for piecewise linear score functions.

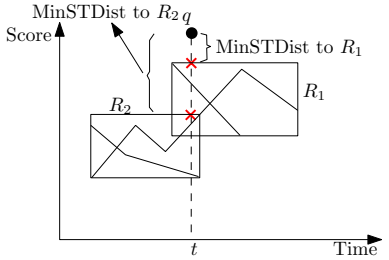


Fig. 3 Querying the R-tree using MinSTDist.

In the literature, various techniques have been proposed under the above framework. They differ in how to split each trajectory and how the R-tree is constructed. Intuitively, if the trajectories are split in a way such that the resulting R-tree MBRs are "nice", then the queries can be processed efficiently. Different metrics [5, 19] have been proposed to measure how "nice" the

MBRs are, e.g., small overlap, small total area, small total perimeter, and various algorithms have been designed to optimize them. However, no metric has been shown to have a provable implication on the query performance. As a result, all these R-tree based indexes remain heuristic in nature, and the query cost could be as high as $\Theta(\frac{N}{B})$ in the worst case. In the next section, we introduce the SEB-tree, a simple index structure for top-$k(t)$ queries, which not only enjoys a strong theoretical guarantee, but also is far more efficient than the state-of-art R-tree based solutions in practice.

2.2 Other related work

Large amounts of efforts have been spent on managing and querying temporal data that come in different forms. Examples include time series data, spatio-temporal trajectories, transactional databases and others. Arguably, many of these works are relevant to this paper. Our review here is brief and only covers the most related ones.

First of all, our framework deals with temporal data when they are represented and stored as piecewise linear functions. A rich body of work has been devoted to approximate time series or spatio-temporal trajectories in this form [5, 27, 32]. For objects in a transactional database, their score functions are staircase functions as we have demonstrated in Section 2. Indexing piecewise linear approximations of time series for the purpose of similarity search has also been explored [12]. However, since the metric for the similarity search is quite different from a ranking query, such an index does not directly help answering ranking queries. The similar reason holds for other index structures (see [14, 37, 41] and references therein) designed for similarity search over time series data. In this case, the query is a temporal object (e.g., a time series), rather than a snapshot query point, and the goal is to retrieve similar temporal objects from the database w.r.t some pre-defined similarity function.

There have been numerous works on indexing historical, spatio-temporal trajectories for various types of spatial and/or temporal queries [4, 9, 11, 16, 19, 30, 33, 34, 36], e.g., range queries, nearest neighbor queries, or even complex pattern queries [18]. As we discussed in Section 2, indeed it is possible to transform a top-$k(t)$ query to a k-nearest neighbor query on spatio-temporal trajectories, and apply the existing techniques for snapshot kNN queries, e.g., the state of the art in [16] using 3DR-tree for two dimensional trajectories. It is also possible to do so with a Multi-Version R-tree (MVR-tree) [19, 39]. However, as noted in [19], MVR-tree is a better choice than R-tree if the goal is support temporal range queries. For snapshot queries, an R-tree based solution usually works well. Indexing structures in transactional databases support queries over objects with staircase curves efficiently [8, 24, 28, 29, 36]. In fact, as indicated in Section 2, we can simply use the MVB-tree to optimally answer the top-$k(t)$ query when all objects could be represented by staircase functions. Unfortunately, this no longer holds for the general setting with piecewise linear functions.

There are also extensive studies on indexing and querying continuous moving objects, e.g., [23, 31, 33, 35, 38, 40]. In numerous cases, the piecewise linear representation is adopted to represent the movements of moving objects and the queries studied are similar to those for spatio-temporal trajectories. An additional requirement in this scenario is to support frequent, dynamic updates at the current time instance efficiently. In particular, to deal with frequent updates in moving objects, Jensen et al. [23] also used the B-tree. However, they used the B-tree to index the Z-values of the movements for moving objects and designed their index specially to handle frequent, dynamic updates. They were mostly interested at queries with a temporal range. Hence, our usage of the B-tree has a fundamentally different purpose. Similar to the case in spatio-temporal trajectories, it is also possible to leverage on these works by transforming the top-$k(t)$ query to a nearest neighbor query. However, since they were not specially designed for ranking queries, special characteristics of the ranking queries were not explored by these approaches. Note that the name SEB-tree has been previously used in the work by Song et al. [38]. It stands for Start/End time stamp B-tree, where the trajectories of the moving objects are partitioned into a set of groups. The segments in each group have the aligned start and end time and these time instances are indexed by the B-tree. Obviously, this SEB-tree enables fast retrieval of partial trajectories that are within a certain time range, but it is fundamentally different from the SEB-tree we have designed and does not help in answering the ranking queries.

Another related query in temporal data is the skyline queries on time series [25]. The objective there is to retrieve skyline time series over a query time interval. A time series belongs to the skyline for a time interval iff it has not been completely dominated by other time series in that time interval. This query was studied for similar motivations as that for the top-$k(t)$ queries. However, the query semantics of the two bears fundamental differences. Hence, the query processing techniques are different as well.

Finally, there exists a large body of research on processing ranking queries in various settings. Interested

readers are referred to the survey by Ilyas et al. [22] on this subject .

3 The SEB-Tree

We present the SEB-tree in this section.

3.1 The design principle

The SEB-tree index for answering the top-$k(t)$ queries leverages on some simple intuitions and randomization.

In the high level, a key observation that motivates our design is that at any time instance t, the result for a top-$k(t)$ query depends on the k "highest" line segments covering t. Furthermore, a line segment from these k "highest" line segments is more likely to belong to the set of k "highest" line segments covering a temporal interval around t. In other words, if we can partition the temporal dimension into disjoint intervals and find the k "highest" line segments w.r.t. each interval, we can index these temporal intervals (e.g., with the t value of the right endpoint of a temporal interval) using an B-tree and store the k "highest" line segments from each interval as data records pointed by each temporal interval at the leaf level of the B-tree. This B-tree, termed as the SEB-tree, obviously provides a highly efficient indexing structure for answering any top-$k(t)$ queries. We next highlight our design principles to help understand how a SEB-tree works and why it works well.

We first break each piecewise linear function into a number of line segments. Now a top-$k(t)$ query is to find the top-k segments from above at a given t. Our construction uses the *upper envelope* of a set of line segments. Specifically, the upper envelope $U(S)$ of a set of segments S is made of the portions of the segments visible from $+\infty$ along the y-axis. Suppose we have sampled a subset of segments S_1 from S, and built its upper envelope $U(S_1)$. An important observation is that given any time instance t, a top-$k(t)$ query only needs to check all the segments from S that locate above $U(S_1)$ at t, if there are at least such k segments. Furthermore, note that $U(S_1)$ is made of disjoint segments (see an example in Figure 4). Hence, for each segment e from $U(S_1)$, we can store the set of segments $C(e)$ from S that ever locate above e within the time extent covered by e. For every e from $U(S_1)$, we refer to the set $C(e)$ as e's *conflict list*. Assume (unrealistically) for now that every conflict list has at least k segments from S. To answer a top-$k(t)$ query, we simply locate the segment e in $U(S_1)$ that spans t, and scan $C(e)$ to find in $C(e)$ the k segments that are the highest at t. The second step is trivial; while the first step can be achieved with a B-tree on the upper envelope $U(S_1)$. Note that $U(S_1)$ consists of segments that do not overlap on the time axis, this becomes a one-dimensional problem and can be easily solved by a B-tree.

Now we need to deal with the assumption that $C(e)$ contains less than k segments or it contains too many. In the former case the query algorithm fails; in the latter case, scanning $C(e)$ will be too expensive. To deal with this issue, the idea of is to use *geometrically decreasing samples*, i.e., we obtain S_1 by sampling with probabilities $1/2, 1/4, 1/8, \cdots$, and build the conflict lists for each of these sampling rates. The intuition is the following. If we sample every segment in S with probability $1/2$, then $C(e)$ should be very small, remembering that a segment in $C(e)$ must be above *all* the sampled segments at some t spanned by e. As we lower the sampling rate, $C(e)$ gets larger. Since we lower the sampling rate by half every time, hopefully one $C(e)$ will have between k and $2k$ segments, i.e., the "right" number of segments that we would like to scan.

Of course to materialize the above intuition several questions need to be formally addressed: Will there always be a $C(e)$ with $\Theta(k)$ segments for any given k (in the expected sense)? How about the total size of the structure? Will it get too large as one segment may potentially belong to multiple conflict lists? We will answer them in Section 3.3 after giving the detailed construction algorithm of the SEB-tree in the next section.

3.2 The index

This subsection contains everything that a practitioner needs to know about how to implement the SEB-tree index for ranking queries on temporal data. We defer the more complicated analysis to the next subsection.

Let S be a set of N segments in the plane. We assume that the segments are in *general position*, i.e., no endpoints of the segments share the same x-coordinates; such an assumption can be easily removed using standard tie-breaking techniques.

The structure. We call a sample from S a *p-sample* if we pick each segment from S randomly with probability p. The basic idea of our index is to draw a collection of p-samples for a series of geometrically decreasing p's; this basic principle has also been successfully applied on numerous geometric range searching problems, though mostly in a theoretical setting. Please see [2,10,13] and the references therein.

We create a series of $\ell + 1$ independent samples of S for $\ell = \lceil \sqrt{\log(N/B)} + \log(k_{\max}/B) \rceil$:

$$S_0, S_1, \ldots, S_\ell,$$

where S_i is a $(1/(2^i B))$-sample of S, $i = 0, 1, \ldots, \ell$. Note that S_i has expected size $N/(2^i B)$.

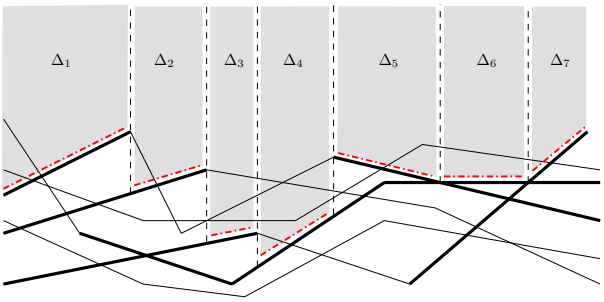


Fig. 4 The trapezoidal decomposition $\mathcal{D}(S_i)$ of the region bounded by the upper envelope U_i of a set of sampled segments S_i. The sampled segments S_i are shown in thickness and their upper envelop U_i is indicated by the dash-dotted lines.

We say that a segment s is *alive* at x if s intersects the vertical line at x; let the coordinates of the intersection point be $(x, s(x))$. If s is not alive at x, we set $s(x) = -\infty$. For each $i = 0, \ldots, \ell$, we compute the *upper envelope* of S_i, defined as the function

$$U_i(x) = \max_{s \in S_i} s(x).$$

Note that U_i is also a piecewise linear function, possibly taking the value $-\infty$ in some intervals. There are well-known algorithms for computing the upper envelope of a set of segments [21], as well as fast and reliable implementations [1]. Consider a particular U_i. From each vertex on U_i, we shoot up a vertical line; if the vertex is an endpoint of a segment, we also shoot down until it hits another segment (or goes to $-\infty$). This results in the *trapezoidal decomposition* of the region bounded by U_i from below. Please refer to Figure 4. We denote by $\mathcal{D}(S_i)$ the trapezoidal decomposition obtained from S_i this way and a trapezoid in $\mathcal{D}(S_i)$ is denoted as Δ. Figure 4 shows the upper envelope (marked by the dash-dotted lines) for a set of sampled segments (shown in thickness). Their trapezoidal decomposition is represented by Δ_1 to Δ_7 as shown in Figure 4.

Consider a trapezoid Δ from some $\mathcal{D}(S_i)$; by convention we will treat Δ as *open*, i.e., it does not include its boundaries. For any segment $s \in S$, we say that s *conflicts* with Δ if s intersects Δ. Then for each segment $s \in S$, we find all the trapezoids that it conflicts with. We will discuss how to do this efficiently in a moment. Note that the sampled segments are treated in the same way.

After this process, each trapezoid Δ can collect all the segments that conflict with it, denoted by $C(\Delta)$. This is called the *conflict list* of Δ. For each level $\mathcal{D}(S_i)$, $i = 0, 1, \ldots, \ell$, letting $\Delta_1, \ldots, \Delta_t$ be the trapezoids in $\mathcal{D}(S_i)$ from left to right, we simply build a B-tree $\mathcal{T}_i$ on the conflict lists $C(\Delta_1), \ldots, C(\Delta_t)$ in order. This means that the indexing attribute of the B-tree is the value of the time instance on the x-axis that corresponds to the left starting point of each trapezoid, and the record in the B-tree is the conflict list of the corresponding trapezoid. If we denote the left x-value for the starting point of a trapezoid Δ_i as $x(\Delta_i)$, then the B-tree essentially indexes the following records, where each record is a (key, record) pair, $\{(x(\Delta_1), C(\Delta_1)), \ldots, (x(\Delta_t), C(\Delta_t))\}$. We built this B-tree for each level, from $\mathcal{D}(S_0)$ to $\mathcal{D}(S_\ell)$, thus we obtain a total of $\ell + 1$ B-trees. These B-trees are our complete index and denoted as the SEB-tree.

The query algorithm. Our query algorithm is also extremely simple as outlined in Algorithm 1. Basically, the query starts with the first level B+ tree and continues to the last level B+ tree if necessary. In any given level i, using a point search with t in the B+ tree, we find the trapezoid $\Delta \in \mathcal{D}(S_i)$ whose x-span contains the query time instance t, which is pointed to by some leaf entry in the B+ tree at this level. We then read the conflict list of Δ, $C(\Delta)$. If $C(\Delta)$ contains more than k segments at t, we return the top-k segments at t with the k highest score attribute values and terminate the search. Otherwise, the search goes to the next level. When we are unable to terminate after iterating through all levels, we simply do a linear scan over the entire S to find the answer.

Algorithm 1: Top-k query at t

```
ℓ0 = ⌈log(k/B)⌉;
for i = ℓ0, ..., ℓ do
    do a point search in T_i and find the trapezoid
    Δ ∈ D(S_i) whose x-span contains t;
    read the conflict list C(Δ);
    let R be the set of segments in C(Δ) alive at t;
    if |R| ≥ k then
        return from R the highest k segments at t;
        terminate;
scan the entire S and return the top-k segments at t;
```

We would like to emphasize that even though the SEB-tree is built based on the envelops for a subset (sampling) of objects at different levels, its query algorithm does return the *exact* results for the top-$k(t)$ queries, due to the last line in Algorithm 1, i.e., it will scan the entire data set if necessary. However, as shown by our theoretical analysis in next section, the query algorithm in most cases terminates before incurring this expensive scan operation and the overall expected query cost is still logarithmic. This is also confirmed by our experimental study in Section 5. The SEB-tree has an excellent query cost and it almost never needs to initiate the scan of the entire data set.

Building the conflict lists. It remains to show how to build all the conflict lists efficiently. Let us focus on a particular level S_i and its trapezoidal decomposition $\mathcal{D}(S_i)$. We will build a hierarchical data structure on

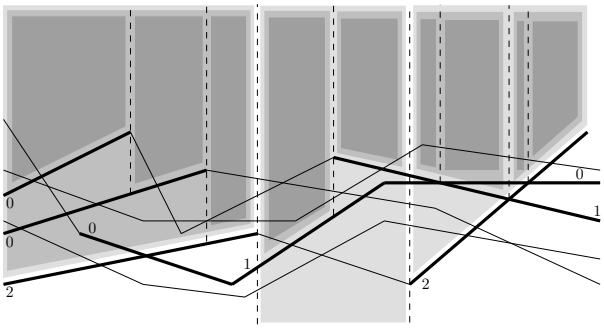


Fig. 5 The hierarchical trapezoidal decomposition imposed by the gradation $L_2 \subseteq L_1 \subseteq L_0$. The two segments numbered with 2 are in L_2 (hence also in L_1 and L_0); the two segments numbered with 1 are in L_1 (hence also in L_0); the four segments numbered with 0 are in L_0. The bottom-level trapezoidal decomposition $\mathcal{D}(L_0)$ consists of the darkest trapezoids; $\mathcal{D}(L_1)$ consists of the darker trapezoids; $\mathcal{D}(L_2)$ consists of the three lighter trapezoids.

top of $\mathcal{D}(S_i)$ such that for any segment $s \in S$, we can find efficiently all the trapezoids $\Delta \in \mathcal{D}(S_i)$ that conflict with s. Let L_0 be the set of segments in S_i. We will build a *gradation* $L_0 \supseteq L_1 \supseteq \cdots \supseteq L_\lambda$, where each L_j is a 1/2-sample of L_{j-1}, $j = 1, \ldots, j = \lambda$. We stop the sampling process when there are only a constant number of segments left. Thus the gradation has $\lambda = O(\log |L_0|)$ levels. For each $L_j, j = 0, 1, \ldots, \lambda$, we build its trapezoidal decomposition $\mathcal{D}(L_j)$. In addition, we further partition each $\mathcal{D}(L_j)$ with the vertical dividing lines from all higher levels $\mathcal{D}(L_{j+1}), \ldots, \mathcal{D}(L_\lambda)$. Please see an example in Figure 5 for the set of sampled segments in Figure 4. Note how some trapezoids in $\mathcal{D}(L_0)$ are further divided by the vertical lines from higher levels, as compared with those in Figure 4. This way obtains a hierarchical trapezoidal decomposition $\mathcal{D}(L_0), \mathcal{D}(L_1), \ldots, \mathcal{D}(L_\lambda)$. We store all the trapezoids in this hierarchy in a tree, where a trapezoid $\Delta \in \mathcal{D}(L_j)$ is a child of some $\Delta' \in \mathcal{D}(L_{j+1})$ if and only if $\Delta \subseteq \Delta'$.

Now, for each segment $s \in S$, we follow this hierarchy recursively, visiting a trapezoid Δ if and only if s intersects with Δ. Eventually we will find all the trapezoids in $\mathcal{D}(L_0)$ that conflict with s. Note that some trapezoids in $\mathcal{D}(S_i)$ may correspond to multiple trapezoids in $\mathcal{D}(L_0)$ due to the further partitioning. In this case s is added to the conflict list of the trapezoid in $\mathcal{D}(S_i)$ only once.

3.3 Analysis

Although our structure of our index is quite simple, its analysis is nontrivial, requiring some abstract notions from computational geometry.

Correctness. We first argue for the correctness of our index, before moving on to the more involved performance analysis. Let $\Delta_0 \in U_0, \Delta_1 \in U_1, \ldots$ be the trapezoids whose x-span contains t from each of the ℓ levels. The query algorithm scans each $C(\Delta_i), i = 0, 1, \ldots$ in order, until we reach some $C(\Delta_i)$ that contains at least k segments alive at t. Suppose we stop at $C(\Delta_i)$. By the definition of the conflict list, any segment not in $C(\Delta_i)$ lies completely outside Δ_i, hence must be lower at t than any segment in $C(\Delta_i)$ alive at t. This proves the correctness of our index.

Space analysis. To bound the total size of our B-tree index, we need to introduce the notion of a *configuration space*, an abstract combinatorial framework with numerous useful geometric instantiations. A *configuration space* is a 4-tuple $\mathcal{X} = (X, \Pi, D, C)$, where X is the *ground set*, Π is set of *configurations*. The two sets are related by two mappings D and C: $\Pi \to 2^X$. Each configuration $\Delta \in \Pi$ is characterized by the two subsets of X: its *defining set* $D(\Delta)$ and *conflict list* $C(\Delta)$. We require that there are at most a constant number of configurations sharing the same $D(\Delta)$; but there could be many with the same conflict list. The *degree* of the configuration space is defined as $d = \max_\Delta |D(\Delta)|$. Given a subset $Y \subseteq X$, the *induced configuration space* $\Pi_{|Y}$ is the set of all configurations Δ such that $D(\Delta) \subseteq Y$ and $C(\Delta) \cap Y = \emptyset$, i.e., all the defining elements of Δ appear in Y but none of the conflicting ones does.

In our setting, the ground set X is the set of N segments in the plane. The configuration space Π includes all the possible top-open trapezoids that are determined by two vertices, each of which can be either an endpoint of a segment in X or the intersection of two segments. The defining set $D(\Delta)$ of a trapezoid Δ is thus the at most four segments that decides its two bottom vertices. So the degree of this configuration space is 4. The conflict list of Δ, as defined before, includes all the segments that intersect Δ. When defined this way, one can easily see that the induced configuration space $\Pi_{|Y}$ for a subset $Y \subseteq X$ of segments exactly consists of all the trapezoids in the trapezoidal decomposition of the region bounded from below by the upper envelope of Y.

We need the following result from [13] with respect to a random sample Y and its induced configuration space $\Pi_{|Y}$.

Lemma 1 (Clarkson and Shor [13]) *Let a configuration space of constant degree be $\mathcal{X} = (X, \Pi, D, C)$ and Y be a p-sample of X. For any configuration $\Delta \in \Pi_{|Y}$:*

$E[|C(\Delta)|] = O(1/p)$.

This lemma essentially states that if we randomly sample a subset of segments S_p from S with a sampling rate p, the conflict list of any segment from $U(S_p)$ contains $O(1/p)$ segments from S on expectation (i.e., the size of the conflict list is bounded). In addition, with the geometrically decreasing sampling rate, the chance

that we have a layer that most conflict lists have more than k segments is high (for typical value of k).

A second technical lemma we need is with respect to the complexity of the upper envelope of n segments in the plane, which is given in [20].

Lemma 2 (Hart and Sharir [20]) *There are $O(n\alpha(n))$ vertices on the upper envelope of n segments in the plane, where $\alpha(n)$ is the inverse Ackermann function.*

Note that $\alpha(n)$ is an extremely slow-growing function. For n as large as $2^{\cdot^{\cdot^{\cdot^{2}}}}\Big\}^{65536 \text{ twos}}$, $\alpha(n)$ is still less than 4. Thus $\alpha(n)$ can be treated as a constant for all imaginable input sizes. Hence, this result guarantees that there are not too many segments from $U(S_p)$ for any sampling rate p for a sampled set of segments S_p from S. This is critical since the number of segments in $U(S_p)$ will decide how many leaf entries we have to index in the $B+$ tree in the SEB-tree.

Because S_i is a sample of expected size $1/2^i \cdot N/B$, the trapezoidal decomposition U_i has expected $O(1/2^i \cdot N/B \cdot \alpha(1/2^i \cdot N/B)) = O(1/2^i \cdot N/B \cdot \alpha(N/B))$ trapezoids. The complexity of U_i is actually $O(1/2^i \cdot N/B \cdot \alpha(N/B))$ with high probability. By Lemma 1, for each $\Delta \in U_i$, we have

$$E[|C(\Delta)|] = O(2^i B).$$

Thus, the expected total size of $\mathcal{T}_i$, which is the total size of all the conflict lists in U_i, is

$$\begin{aligned}\textstyle\sum_{\Delta \in U_i} E[|C(\Delta)|] &= O(1/2^i \cdot N/B \cdot \alpha(N/B)) \cdot O(2^i B) \\ &= O(N\alpha(N/B)).\end{aligned}$$

Since the whole index consists of $\ell+1 = O(\sqrt{\log(N/B)}+ \log(k_{\max}/B))$ B-trees, we obtain the following.

Corollary 1 *Our index occupies the following number of disk blocks in expectation:*

$$O\left(\frac{N}{B}\alpha\left(\frac{N}{B}\right)\left(\sqrt{\log\frac{N}{B}} + \log\frac{k_{\max}}{B}\right)\right).$$

Query performance analysis. We now analyze the query performance of our index. The analysis depends on the following key lemma.

Lemma 3 *The probability that the algorithm 1 queries $\mathcal{T}_i$ is at most $2^{-\Omega((i-\ell_0)^2)}$, for any $i > \ell_0 = \lceil \log(k/B) \rceil$.*

Proof We first consider the probability that if the algorithm queries $\mathcal{T}_i$, what is the probability that it will fail. Let Δ be the trapezoid in U_i whose x-span contains t. The query algorithm fails on $\mathcal{T}_i$ because there are less than k segments in $C(\Delta)$ that are alive at t. For this to happen, at least one of the top-k segments at t must have been sampled into S_i. Otherwise, these k segments will all conflict with Δ, leading to a contradiction. Thus, the probability that at least one of the top-k segments is sampled is an upper bound on the probability that $\mathcal{T}_i$ fails. Since S_i is a $(1/(2^i B))$-sample of S, this probability is bounded by

$$1 - \left(1 - \frac{1}{2^i B}\right)^k \le 1 - \left(1 - \frac{k}{2^i B}\right) = \frac{k}{2^i B}.$$

Now we consider the probability that the algorithm queries $\mathcal{T}_i$, which happens if and only if all of $\mathcal{T}_{\ell_0}, \mathcal{T}_{\ell_0+1}, \ldots, \mathcal{T}_{i-1}$ have failed. Since the B-trees are constructed from independent samples, this happens with probability at most:

$$\begin{aligned}&\frac{k}{2^{\ell_0}B} \cdot \frac{k}{2^{\ell_0+1}B} \cdot \cdots\cdot \frac{k}{2^{i-1}B} \\ &= \left(\frac{k}{2^{\ell_0}B}\right)^{i-\ell_0} \left(\frac{1}{2^0} \cdot \frac{1}{2^1} \cdot \cdots\cdot \frac{1}{2^{i-1-\ell_0}}\right) \\ &\le \frac{1}{2^0} \cdot \frac{1}{2^1} \cdot \cdots\cdot \frac{1}{2^{i-1-\ell_0}} \\ &= 2^{-\Omega((i-\ell_0)^2)}.\end{aligned}$$

□

The algorithm always visits $\mathcal{T}_{\ell_0}$. The expected cost, in terms of the number of I/Os, is given by:

$$\begin{aligned}O(\log_B N + E[|C(\Delta)|/B] &= O(\log_B N + 2^{\ell_0} B/B) \\ &= O(\log_B N + 2^{\ell_0}) \\ &= O(\log_B N + k/B).\end{aligned}$$

Next consider the B-tree $\mathcal{T}_i, i > \ell_0$. Conditioned on the event that we query $\mathcal{T}_i$, the expected I/O cost on $\mathcal{T}_i$ is $O(\log_B N + E[|C(\Delta)|]/B) = O(\log_B N + 2^i)$. By Lemma 3, we query $\mathcal{T}_i$ with probability $2^{-\Omega((i-\ell_0)^2)}$, so the (unconditioned) expected cost on $\mathcal{T}_i$ is

$$\begin{aligned}&2^{-\Omega((i-\ell_0)^2)} \cdot O(\log_B N + 2^i) \\ &\le 2^{-\Omega((i-\ell_0)^2)} \log_B N + 2^{-\Omega((i-\ell_0)^2)} \cdot 2^{i-\ell_0} k/B \\ &\le 2^{-\Omega((i-\ell_0)^2)} \log_B N + 2^{-\Omega((i-\ell_0)^2)} \cdot k/B.\end{aligned}$$

By the linearity of expectation, the total expected cost for all the trees $\mathcal{T}_{\ell_0+1}, \ldots, \mathcal{T}_\ell$ is $O(\log_B N + k/B)$ I/Os, as the series $2^{-\Omega((i-\ell_0)^2)}$ decreases faster than a geometric series.

Finally, we need to account for the expected cost of the algorithm falling into the brute-force one on the last line, which leads to $O(N/B)$ I/Os. However, this happens only with probability

$$\begin{aligned}2^{-\Omega((\ell+1-\ell_0)^2)} &= 2^{-\Omega((\sqrt{\log(N/B)}+\log(k_{\max}/B)-\log(k/B))^2)} \\ &\le 2^{-\Omega((\sqrt{\log(N/B)})^2)} \\ &= 2^{-\Omega(\log(N/B))} \\ &\le (B/N)^{O(1)}.\end{aligned}$$

Thus this only adds $O(1)$ to the overall expected query cost, as long as we set the constant before ℓ properly, which naturally leads to the following result:

Theorem 1 *For any query at t, our index finds the top-k segments at t in expected $O(\log_B N + k/B)$ I/Os.*

Construction time analysis. Finally we analyze the time required to build our index. Generating all the samples $S_0, S_1, \ldots, S_\ell$ takes $O(N)$ time. Building the upper envelopes for all of them takes $O(N/B \log(N/B))$ time using the algorithm of [21] and because the sizes of the samples form a geometric series. Once we have all the conflict lists, building all the B-trees takes time proportional to the total size of our index, i.e., $O(N\alpha(N/B)(\sqrt{\log(N/B)}+\log(k_{\max}/B)))$, so we will concentrate on analyzing the cost of constructing all the conflict lists.

Focusing on a particular level S_i. Recall that to build the conflict lists $C(\Delta)$ for all $\Delta \in \mathcal{D}(S_i)$, we construct a gradation $S_i = L_0 \supseteq L_1 \supseteq \cdots \supseteq L_\lambda$ and a corresponding hierarchical trapezoidal decomposition. Since for each segment s, we visit all and only those trapezoids in this hierarchy that conflict with s, the total time spent will be proportional to

$$\sum_{j=0}^{\lambda} \left(\sum_{\Delta \in \mathcal{D}(L_j)} |C(\Delta)| \right).$$

Let us bound the total conflict list size for $\mathcal{D}(L_j)$. Recall that $\mathcal{D}(L_j)$ is not precisely the trapezoidal decomposition of the upper envelope of L_j; it is slightly finer as we further divide it using the vertical lines from all higher levels. However, since the sizes of levels geometrically decrease, the further partitioning does not increase the complexity of $\mathcal{D}(L_j)$ by more than a constant factor. Thus there are still $O(|L_j|\alpha(|L_j|))$ trapezoids in $\mathcal{D}(L_j)$. Since L_j is a $1/2^j$-sample of S_i, which is a $1/(2^iB)$-sample of S, L_j is effectively a $1/(2^{i+j}B)$ sample of S. Invoking Lemma 1, we have $E[|C(\Delta)|] = 2^{i+j}B$ for any $\Delta \in \mathcal{D}(L_j)$. Also $|L_j|$ is $O(N/(2^{i+j}B))$ with high probability, thus all the conflict lists of $\mathcal{D}(L_j)$ has an expected total size of $O(N\alpha(N/B))$. Summed over all $\lambda = O(\log(N/B))$ levels in the gradation, the total cost for S_i is $O(N\alpha(N/B)\log(N/B))$. Since we have $\sqrt{\log(N/B)} + \log(k_{\max}/B)$ B-trees, the total time spent to build our entire index is $O(N\alpha(N/B)\log(N/B)(\sqrt{\log(N/B)} + \log(k_{\max}/B)))$.

Theorem 2 *Our index can be built in expected time*

$$O\left(N\alpha\left(\frac{N}{B}\right)\log\frac{N}{B}\left(\sqrt{\log\frac{N}{B}} + \log\frac{k_{\max}}{B}\right)\right).$$

3.4 Updating the SEB-tree

The SEB-tree can also be efficiently updated without affecting the space and query guarantees. Since the SEB-tree is nothing but a collection of B-trees, in the sequel we will just describe how to update the level-i tree; the same procedure is applied on all the B-trees.

Recall that in the level-i B-tree, we take a $(1/(2^iB))$-sample S_i of all the segments, build the trapezoidal decomposition $\mathcal{D}(S_i)$ of the upper envelope of S_i, and then store the conflict lists of all the trapezoids in $\mathcal{D}(S_i)$ in a B-tree. When the segment s to be inserted or deleted is not one of the sampled segment, the procedure is extremely simple. In fact, it is the same as how the conflict lists were built in the construction algorithm of Section 3.2. More precisely, we first follow the hierarchical trapezoidal decomposition $\mathcal{D}(L_0), \ldots, \mathcal{D}(L_\lambda)$ to find all the conflict lists where s belongs to, and then simply insert or delete s from these lists. Since each segment has probability only $1/(2^iB)$ to be sampled into S_i, this simple update procedure is invoked most of the time. In the rare case where s is being sampled, we first need to restructure the upper envelope of S_i and the hierarchical trapezoidal decomposition before updating the conflict lists. This procedure is more complicated, but fortunately it only happens for about $1/(2^iB)$ of the insertions and deletions. Below we give the details for each of the steps. We will focus on insertions only; deletions can be handled similarly.

The first component is to update the upper envelope and the hierarchical trapezoidal decomposition. When s is inserted into S_i, the upper envelope of S_i (hence $\mathcal{D}(S_i)$) might get changed. To update $\mathcal{D}(S_i)$, we simply reconstruct the portion of the upper envelope in the x-span of s using the algorithm of [21]. Similarly we update $\mathcal{D}(L_1), \ldots, \mathcal{D}(L_\lambda)$ for each level in the hierarchical trapezoidal decomposition on top of $\mathcal{D}(S_i)$.

Next, we need to update the conflict lists. For all the trapezoids Δ in $\mathcal{D}(S_i)$ that no longer exist in the new $\mathcal{D}(S_i)$, we collect their conflict lists $C(\Delta)$ after removing the duplicates. Then we again, as in the construction algorithm, filter each segment down the hierarchical trapezoidal decomposition to find its new trapezoids in conflict. Finally, we add s to the conflict lists that it belongs to.

3.5 Comparison of different solutions

We have summarized the theoretical upper bounds of the SEB-tree comparing to the other baseline solutions for solving top-$k(t)$ queries in Figure 6. Note that the SEB-tree is a randomized structure while the R-tree and the MVB-tree solutions are deterministic, the bounds for the SEB-tree are the expected case while they are

	SEB-tree	R-tree	MVB-tree
index size	$O(\frac{N}{B}\alpha(\frac{N}{B})(\sqrt{\log \frac{N}{B}} + \log \frac{k_{\max}}{B}))$	$O(\frac{N}{B})$	$O(\frac{N^2}{B})$
construction cost	$O(N\alpha(\frac{N}{B}) \log \frac{N}{B}(\sqrt{\log \frac{N}{B}} + \log \frac{k_{\max}}{B}))$	$O(\frac{N}{B} \log \frac{N}{B})$	$O(\frac{N^2}{B} \log \frac{N}{B})$
query cost	$O(\log_B \frac{N}{B} + \frac{k}{B})$	$O(\frac{N}{B})$	$O(\log_B \frac{N}{B} + \frac{k}{B})$
update cost	unknown	$O(\log_B \frac{N}{B})$	not supported

Fig. 6 Theoretical upper bounds of the SEB-tree (expected), R-tree and MVB-tree (worst case) for top-$k(t)$ queries.

worst-case bounds for the other two. However we emphasize that, as with any randomized algorithm, the expectation is with respect to the randomization within the structure itself, not the input data. It should be pointed out that it is possible the query cost could be as high as $O(N/B)$ for the SEB-tree, but this happens extremely rarely (guaranteed by Lemma 3, as well as observed in our experiments). More importantly, whether this happens or not depends on the internal randomization within the structure, *not* on the input data or query. On the other hand, the two deterministic solutions will *always* attain their worst-case bounds on certain inputs. Nevertheless, in situations where the performance needs to be absolutely guaranteed, randomized algorithms should be avoided at all. Unfortunately for the top-$k(t)$ problem there are no deterministic solutions with good worst-case guarantees on both the index size (hence construction cost) and query time.

4 Extensions

We discuss a space-query tradeoff of the SEB-tree and the generalization of the SEB-tree to support aggregate score functions over multiple attributes in this section.

4.1 Space-query tradeoff of the SEB-tree

The SEB-tree uses near-linear space (only with a small logarithmic factor). This introduces little storage overhead that does not raise a concern for most practical applications. In practice, one may further reduce the space consumption of the SEB-tree without abruptly changing its query performance. In fact, there exists a simple method to gradually adjust the behavior of the SEB-tree to achieve a nice tradeoff between its query cost and space consumption.

Our idea is built upon the observation given by Lemma 1. Essentially, one expects to see $O(1/p)$ conflicting segments for any one trapezoid where p is the sampling rate for some level S_i. For level $i \in [0, \ell]$, $p = \frac{1}{2^i B}$. Now assume that a query has come to a particular trapezoid Δ in level i, directed by the search in the ith level B-tree. If the size of Δ's conflict list, $|C(\Delta)|$, is way larger than the expectation, $2^i B$, say by a factor of λ, we know that this is an outlier and we should try to avoid. In that case, we may choose to skip reading the conflict list of this particular Δ. Instead, the query continues to the B-tree on the next level. The intuition is that the probability for a query to consistently encounter such bad cases across multiple levels is extremely low in practice. Of course, if the query is currently at the last level, we do not skip reading any conflict list, since this represents the last line of defense to avoid the (extremely rare) expensive linear scan of the entire database.

By doing the above on the query side, an immediate optimization for reducing the size of the SEB-tree is to simply throw away the conflict lists for those trapezoids Δ at level i that have $|C(\Delta)| > \lambda \cdot 2^i B$. Obviously, there is a tradeoff between the query efficiency and the space consumption for different values of λ. We denote a SEB-tree built by applying the above idea with a particular value for λ as the SEB-*tree*λ. Clearly, the basic version of the SEB-tree is equivalent to the case when $\lambda = \infty$, i.e., keeping conflict lists for all trapezoids. Our experiments explore this interesting tradeoff in more details and reveal that typically setting $\lambda = 3$ or 4 will reduce the size of the SEB-tree by a factor of 1/2 or 1/3, while keeping the query cost close enough to the optimal performance given by the basic version of the SEB-tree.

4.2 Aggregation of multiple attributes

In some situations, the objects could have multiple temporal attributes, say $a_1(t), \ldots, a_d(t)$, where each $a_i(t)$ is a piecewise linear function, and a top-$k(t)$ query asks for the top-k objects at time t according to an aggregation function $f(a_1(t), \ldots, a_d(t))$, e.g., sum, max, or min. Next, we discuss how to extend the SEB-tree to support top-$k(t)$ queries using such aggregation functions.

Predefined aggregation functions. The extension is straightforward when the aggregation function f is predefined. All we need to do is to compute the aggregation function $f(t) = f(a_1(t), \ldots, a_d(t))$ and then build the SEB-tree on these $f(t)$'s. The SEB-tree works directly for most aggregation functions, such as sum, average, max, min, since if the $a_i(t)$'s are piecewise linear, $f(t)$ is also a piecewise linear function. Thus the construction, query, and update algorithms remain the same, although the complexity of the SEB-tree might

change slightly: When the aggregation function is sum or average, it is clear that the total size (total number of pieces) in an $f(t)$ is the same as that of the $a_i(t)$'s, since each turning point in an $a_i(t)$ contributes to at most one turning point in $f(t)$. So all the bounds remain the same in this case. When the aggregation function is max or min, $f(t)$ is the upper or lower envelope of these $a_i(t)$'s (not to be confused with the upper envelopes we used in the construction of the SEB-tree), so the size of $f(t)$ could be more than linear by an $O(\alpha(N))$ factor by Lemma 2. So all bounds for the SEB-tree hold after replacing N with $N\alpha(N)$. But for all practical values of N, this increase is negligible.

In fact, even if the aggregation function is a higher-degree polynomial, say $f(t) = a_1^2(t) + \cdots + a_d^2(t)$, the SEB-tree still works. In this case, $f(t)$ is a piecewise quadratic function, but the upper envelopes and conflict lists can still be defined in the same way as before, and the SEB-tree works in exactly the same way as before. The analysis also stays the same, except that the bound in Lemma 2 becomes $O(n2^{\alpha(n)})$ [3]. This in turns changes N to $N2^{\alpha(N)}$ in all subsequent bounds. Note that although $2^{\alpha(N)}$ looks "exponential", it is not. In fact it is still close to a constant due to the extremely slow growth of the inverse Ackermann function.

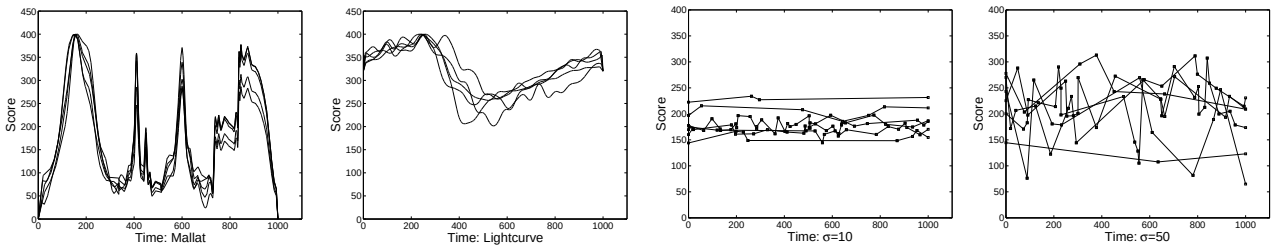
Ad hoc aggregation functions. When the aggregation function is only given at query time, a standard technique is to build a SEB-tree on each attribute $a_i(t)$ separately, and then merge the results together using the *threshold algorithm (TA)* [15] at query time. Note that TA requires sorted accesses to each ranked list, but the SEB-tree supports such accesses easily by querying $\mathcal{T}_0, \mathcal{T}_1, \ldots$ in order. In this case the index size and construction costs of the SEB-trees remain the same as before, but we do not have a bound on the query cost any more, because in order to find the top-k objects at time t, we may need to access more than k objects from each SEB-tree. Nevertheless, since TA is instance-optimal, it is guaranteed that we will not retrieve more objects than necessary from each SEB-tree. However, we note that this optimality holds only when we restrict ourselves to the framework in which TA is analyzed in [15], i.e., the algorithm cannot access an object before it is accessed in sorted order on at least one of its attributes.

5 Experiments

We implemented the SEB-tree using C++. We uses the CGAL library [1] for computing the upper envelope of a set of line segments, and the TPIE library [7] for disk-based B-trees. To compare against the baseline solution, we obtained the latest spatial index library from [17] that includes the R*-tree, the MVR-tree (multi-version R-tree, the generalization of the MVB-tree to R-trees) and the TPR-tree (time parametrized R-tree) [35, 40]. Our observation is that, for snapshot queries, the R*-tree outperforms the MVR-tree and the TPR-tree. The latter ones are preferred for temporal range queries. The same observation has also been observed by several latest works in indexing spatio-temporal trajectories [16, 19]. Hence, we compared the SEB-tree against the baseline solution using the two-dimensional R*-tree from [17]. In addition, we also tried to optimize the segmentation of each object before indexing them by the R*-tree. There are several methods in the literature. We choose to use the global distance-based segmentation approach [5]. In the sequel, we simply refer this baseline approach as the *R-tree*. Lastly, we have also implemented the MVB-tree based solution. Our MVB-tree implementation was based on the TPIE library, following the guidelines provided in [6]. Note that there is a small cost, associated with finding the root of the MVB-tree during its query process. All experiments were executed on a 64-bit Linux machine with 4GB of RAM and a 2GHz Intel Xeon(R) CPU.

Data sets. We have run extensive experiments on both real and synthetic data sets. A sample of five objects in each data set for some of the data sets used in our experiments are shown in Figure 7. Note that, here, each object is simply a function as discussed in the formal problem definition in Section 1. For real data sets, we used the time series data from Keogh's CD ROM [26] and applied the SWAB method [27] to transform them into piecewise linear line segments. In particular, we chose the *Mallat Techno-metrics* (*Mallat* in short) and the *Lightcurve* data sets (see Figure 7(a) and 7(b)), the two largest data sets in [26] in terms of the number of objects. The *Mallat* data set contains 2400 time series and each has 1024 time instances. The *Lightcurve* data set contains 5000 time series and each has 1024 time instances. We used the SWAB method to represent each time series with 200 line segments.

The real data sets are too small to test the scalability of the indexes, so we also generated a collection of synthetic data sets, with which we can also control various important characteristics. We first pick m, the average number of segments the score functions. Then for each object's score function f_i, we first randomly pick a mean value $\mu_i \in [100, 300]$, as well as a random integer m_i from $[1, 2m]$ as the number of segments it contains. Next, we randomly generate m_i time instances $t_1 < t_2 < \cdots < t_{m_i}$ from $[0, 1000]$ along the time dimension. After that, for each t_j, we set $f_i(t_j)$ randomly following the normal distribution $N(\mu_i, \sigma^2)$ for



(a) Mallat, $m = 200$. (b) Lightcurve, $m = 200$. (c) Synthetic: $\sigma = 10$. (d) Synthetic: $\sigma = 50$.

Fig. 7 Random samples of our data sets: a sample of 5 objects.

some predetermined standard deviation σ. If $f_i(t_j)$ gets out of the domain $[0, 400]$, we regenerate a new $f_i(t_j)$. In the last step, we simply connect $(t_{j-1}, f_i(t_{j-1}))$ to $(t_j, f_i(t_j))$ for all $i \in [2, m]$ by a line segment. Clearly, this way allows us to flexibly adjust the degree of intersections among the objects, as well as the stableness for the distribution of the top-k at different time instances. For example, as shown in Figure 7(c) and 7(d), smaller σ values lead to fewer intersections and larger σ values result in a significant number of crossovers.

Setup. We explored the main characteristics of the SEB-tree and its variations, compared to the baseline approach. The metrics include index size, construction cost, query cost, and update cost. In particular, we will investigate how various data characteristics, such as σ, n (total number of objects), and m (the average number of segments in each f_i), may affect the behaviors of different indexes. Note that the total number of line segments for all objects in the database, N, is roughly nm. For the SEB-tree index, we also studied the SEB-treeλ that improves its space usage in practice. In particular, we set $\lambda = 3$ and $\lambda = 4$ since these two values provide a nice tradeoff between space consumption and query cost, as indicated by our experiments. Unless otherwise specified, the default values for various parameters are: n =10,000, $m = 500$, $\sigma = 30$, $k = 100$, $k_{\max} = 200$. Hence, by default the size of the database (N) is 5 million line segments. Each line segment is represented by four doubles, and two integers for its object id and segment id within that object, i.e., 40 bytes in total.

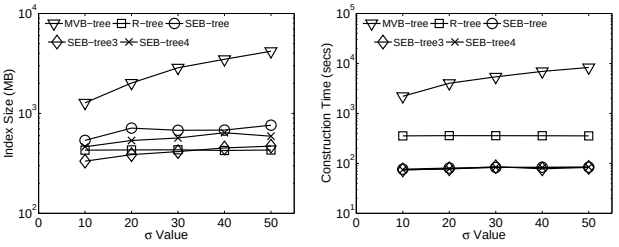
For all experimental figures, except otherwise specified, the index size, the construction cost and the query cost are all shown in log scale. Each query cost is obtained by averaging over 10,000 queries at random time instances. For most results on the query cost, we choose to report only the I/O cost, as the query time is always proportional to the I/O cost. The two real data sets give fairly similar results in all experiments. Hence, we only report the results from the lager data set *Lightcurve*.

5.1 Index size and construction cost

In the first set of experiments, we study the index size and construction cost of various indexes for changing values of σ, n and m. For the initial construction of the index, we always bulkload the R-tree as well as the B-trees within the SEB-tree. The MVB-tree is constructed using its insertion algorithm. The fill-factor is set to be 90% for the R-tree, the MVB-tree and all the B-trees. We did *not* include the cost of finding optimal segmentation before indexing them using the R-tree in the reported times, since this is an optional step. Thus we are giving full advantages to the R-tree: The construction time is the minimum possible (it is just the time to bulkload the R-tree) while we are assuming that the R-tree's quality has been optimized by this optional step. The construction time of the MVB-tree based solution includes two parts: the time to find all intersecting vertices and the time to build the MVB-tree afterward.

The size of the SEB-tree and its variants, the SEB-treeλ, in general increases with the variance in the database gets larger, as shown in Figure 8(a). This is because as the segments get more "vertical", they are likely to intersect more trapezoids in $\mathcal{D}(S_i)$, and hence are stored in more conflict lists. The R-tree's size is not affected by σ since the size of the data set remains as a constant, which is obvious. Note that it is well known that R-tree takes linear space in terms of the size of the input data set. For example, in this experiment, each line segment takes four doubles (starting and ending points in the 2D space) and two integers (the object and line segment ids) to represent, for 5 million line segments the size of the input data set is approximately 200MB (R-tree takes more space in practice due to the index level nodes, but theoretically its size is $O(N)$). From the figure we see that SEB-tree does take more space than the R-tree (as well as the input data set), but not by much. Even in the case when $\sigma = 50$ (this is a considerably large variance for the domain of $[0, 400]$), the size of the SEB-tree is still well within twice the R-tree size. As expected, SEB-tree3 and SEB-tree4 reduce the size of the SEB-tree, and in particular, the size of SEB-tree3 is comparable to that of the R-tree. SEB-tree4 achieves a size reduction of 1/3 in most cases. The bumps in the size curves are due to the fact that the SEB-trees are randomized structures. Lastly, the size of the MVB-tree is significantly larger (almost one order

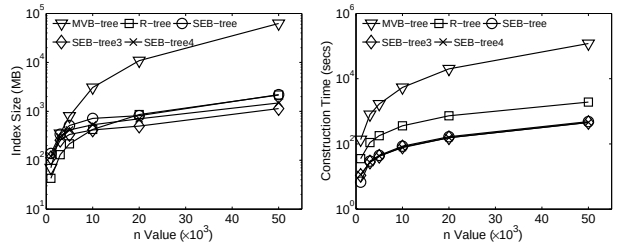
of magnitude) than both the R-tree and the SEB-trees, which is due to the fact that it has to store significantly more segments produced by any intersection from the input segments. Its size also increases quickly when σ becomes larger, as larger variance means that more intersecting vertices will be produced by input segments. This means that the MVB-tree solution scales poorly, compared to the SEB-trees.



(a) Index size. (b) Construction time.

Fig. 8 Effect of σ on index size and construction time, n =10,000, $m = 500$, $N = n \times m$.

The construction of the SEB-trees is much faster that for the R-tree and the MVB-tree solution, as seen in Figure 8(b), by almost 8 times and two orders of magnitude respectively. The construction cost of the SEB-tree increases with σ, for similar reasons as explained above, but the increase is small. In contrast, the construction cost of the MVB-tree solution increases significantly as σ for two reasons. Firstly, a larger σ value implies more intersecting segments and calculating all intersecting vertices becomes more expensive. Secondly, more intersecting vertices further leads to a higher number of insertion operations into the MVB-tree.

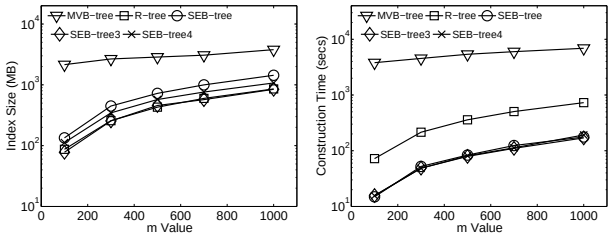


(a) Index size. (b) Construction time.

Fig. 9 Effect of n on index size and construction time, $m = 500$, $\sigma = 30$, $N = n \times m$.

We next study the scalability of the SEB-trees as n, the number of objects in the database, gets larger. The results are shown in Figure 9. The R-tree's size is strictly linear, which is again obvious. On the other hand, the MVB-tree's size is linear to the number segments produced by intersecting all input segments, which is much larger than N (in the worse case it could be $O(N^2)$), as indicated in Figure 9(a). However, the trend of the SEB-tree's size is quite interesting. Theoretically speaking, Corollary 1 indicates that it should also be almost linear, but the figure suggests a sublinear trend at first and then becomes linear later. This interesting trend is the result of the interplay between two opposite forces. As n gets larger, the SEB-tree will sample more segments, which will push their upper envelope higher and then reduce the size of the trapezoids in $\mathcal{D}(S_i)$. On the other hand, as more segments are sampled, their envelope also tends to have more vertices, resulting in more trapezoids. Thus, a larger n leads to more conflict lists (one for each trapezoid) but each list tends to be smaller. This interplay gives rise to the interesting trend we are observing in Figure 9(a), which is not captured by the worst-case analysis of Corollary 1. From the experiments, we see that the SEB-tree's size is actually better than the bound in Corollary 1. For n large enough, the SEB-tree's size is almost the same as the R-tree, with the SEB-tree3 and SEB-tree4 being even smaller. The size of the MVB-tree becomes two orders of magnitude larger than both the R-tree and the SEB-trees when n goes beyond 40,000.

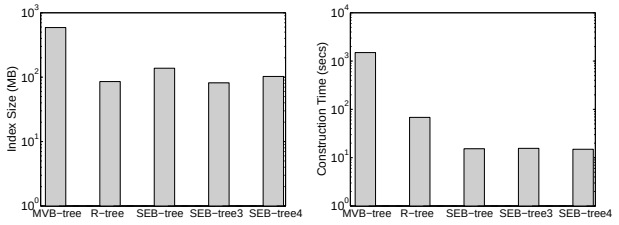
As far as the construction cost is concerned w.r.t. n, Figure 9(b) indicates that the SEB-trees are much cheaper to build than the R-tree and the MVB-tree solutions, by 5 to 8 times and two orders of magnitude respectively, as also observed previously in Figure 8(b).



(a) Index size. (b) Construction time.

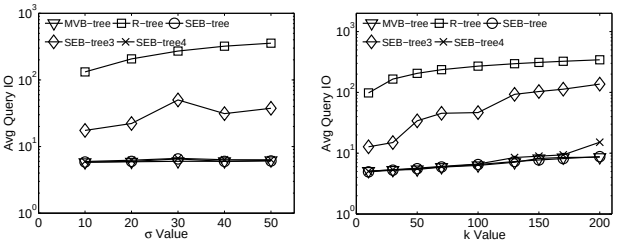
Fig. 10 Effect of m on index size and construction time, n =10,000, $\sigma = 30$, $N = n \times m$.

Finally, we study the effect of m, the average number of segments per object, on the index size and construction cost, while fixing n =10,000. Figure 10 shows the results. In this case, both the size and construction cost are almost linear for the R-tree and the SEB-trees, following the theoretical analysis. Here the interplay mentioned above for the SEB-trees does not exist because the effect of m is only to increase the scale of the data horizontally: The number of sampled segments at any particular time instance is not affected by m. The MVB-tree's size and construction cost could be quadratic to m (since $N = mn$) in the worst case, and they increase with m in practice. We observe that its size and construction cost are significantly higher than both the R-tree and SEB-trees as shown in Figure 10(a) and Figure 10(b), by one to two orders of magnitude. Figure 10(a) shows that the SEB-tree is larger than the R-tree, which agrees with the n =10,000 point in Fig-



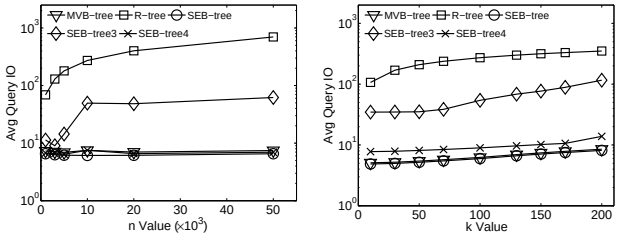
(a) Index size. (b) Construction time.

Fig. 11 Results on Lightcurve, n =5000, $m = 200$, $N = n \times m$.



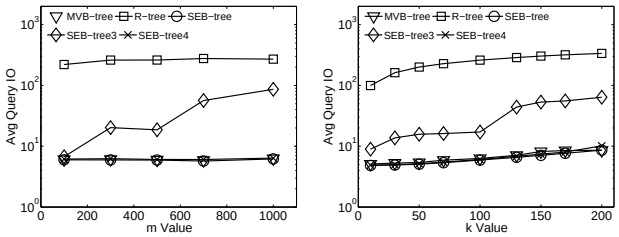
(a) IO: $k = 100$, vary σ. (b) IO: $\sigma = 30$, vary k.

Fig. 12 Effect of σ on query: n =10,000, $m = 500$, $N = n \times m$.



(a) IO: $k = 100$, vary n. (b) IO: n =10,000, vary k.

Fig. 13 Effect of n on query: $m = 500$, $\sigma = 30$, $N = n \times m$.



(a) IO: $k = 100$, vary m. (b) IO: $m = 500$, vary k.

Fig. 14 Effect of m on query: n =10,000, $\sigma = 30$, $N = n \times m$.

ure 9(a). However, note also that the n =10,000 point is actually the worst case for the SEB-tree as seen in Figure 9(a); the SEB-tree's size gets better for larger n. Once again, the SEB-trees have significantly better construction time than the R-tree (Figure 10(b)).

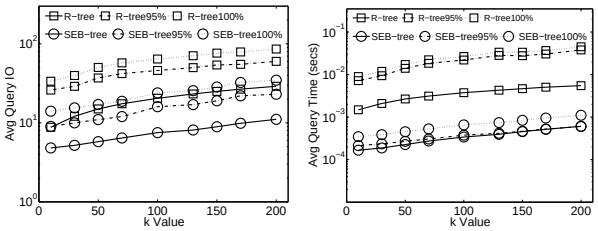
We also investigated the index size and construction cost over real data sets. Figure 11(a) and 11(b) show that the SEB-trees have comparable sizes to the R-tree and much better (almost one order of magnitude better) construction cost. Both of them clearly outperform the MVB-tree solution. In terms of the index size, both of them are roughly one order of magnitude smaller than the MVB-tree solution. In terms of the construction cost, the R-tree solution is one order of magnitude more efficient and the SEB-trees are two orders of magnitude more efficient, than the MVB-tree solution.

These results indicate that the MVB-tree solution is not practical for applications with large data sets, due to its extremely poor scalability. The only viable alternatives are the SEB-trees and the R-tree solution.

5.2 Query cost

We next shift our attention to the query cost, a fundamental metric of any index structure. Even though previous results have already exclude the practicality of the MVB-tree solution, for completeness, we still include its query performance in the comparison.

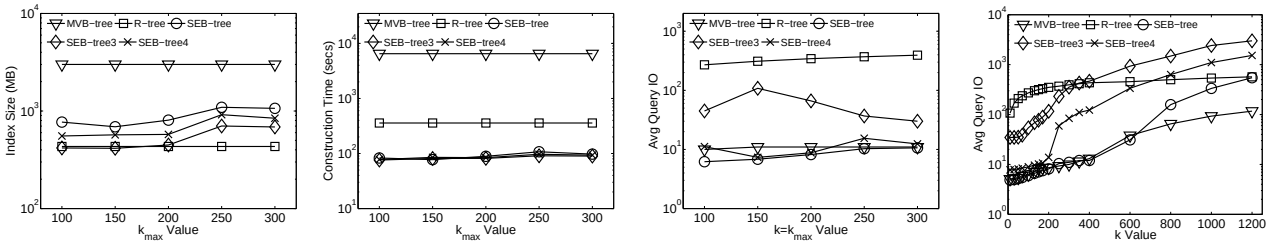
The query cost for varying σ, n and m are shown in Figures 12, 13 and 14 respectively. Within each setting, we also tested an experiment for varying k while fixing σ, n and m. A general trend shown from all these experiments is that the SEB-tree indexes are faster than the R-tree by usually more than an order of magnitude, and have similar query performance as the MVB-tree solution. In particular, the performance of the SEB-tree4 is almost indistinguishable from that of the SEB-tree, making it an ideal candidate for reducing the space consumption (by as much as 1/3). The performance of the SEB-tree3 lies in between of the SEB-tree and the SEB-tree4, but is still clearly better than the R-tree. These experiments confirm the superior query efficiency of the SEB-tree. For example, for n =50,000 and $m = 500$ in Figure 13(a), i.e., 25 million line segments in the database, the SEB-tree takes less than 10 I/Os to answer a top-$k(t)$ query for $k = 100$, and the MVB-tree takes about 15 I/Os, while the R-tree has to take close to 1000 I/Os. Another nice feature of the SEB-tree is that the effect of increasing any of σ, n, m, or k on the query performance is really small, while they have a relatively larger impact on the R-tree. Figures 12, 13 and 14 only show the query I/O cost; the query time is always proportional to the I/O cost.



(a) Query IO. (b) Query time.

Fig. 15 Query on Lightcurve, n =5000, $m = 200$, $N = n \times m$.

Since the SEB-tree is a randomized structure, and Theorem 1 just bounds the expected query cost. One concern is that query cost may have a large variance. In the experiments, we found that the query cost distribution is actually concentrated around the average. To illustrate this point clearly, we used the real data set *Lightcurve*. In Figures 15(a) and 15(b), we plot both



(a) Index size. (b) Construction time. (c) Query IO ($k = k_{\max}$). (d) Query IO: $k_{\max}$=200 vary k.

Fig. 16 Effect of $k_{\max}$: n =10,000, $m = 500$, $\sigma = 30$, $N = n \times m$.

the average query cost curve, the 95% curve, i.e., 95% of the queries have a cost lower than this curve, and the 100% curve (i.e., the worst case query cost), for the R-tree and SEB-trees. We omitted the MVB-tree solution from this Figure, since it has similar query performance as the SEB-tree and its construction cost and index size are orders of magnitude worse. From these results, we see that the 95% curve for the SEB-tree is still better than the average cost of the R-tree, and the worst case curve (100% curve) for the SEB-tree becomes slightly worse the average cost of the R-tree, but both of these curves are still an order of magnitude better than the corresponding 95% and 100% curves of the R-tree.

5.3 Effect of Kmax

Recall from Section 3.2, the construction of the SEB-tree depends on the value of $k_{\max}$. So far, we have set the $k_{\max}$ to its default value 200. In this section, we explore the impact of $k_{\max}$ on the performance of the SEB-tree and its variants.

Specifically, we first vary $k_{\max}$ from 100 to 300 and build the SEB-trees correspondingly. The results are shown in Figures 16(a), 16(b) and 16(c). As far as the index size is concerned, larger $k_{\max}$ value in general will increase the space requirement for SEB-trees: One more B-tree is needed whenever $k_{\max}/B$ doubles. This means that before a small number of levels has been added, $k_{\max}$ will reach an extremely large value, .i.e., for typical $k_{\max}$ values, this will not introduce a large overhead. And the size of SEB-trees is still well within the comparable range to the size of the R-tree and one order of magnitude smaller than the MVB-tree, as indicated by Figure 16(a). For example, in most real-world applications, ranking queries typically ask for a small number of objects in the underlying database. In fact, most existing works on top-k or kNN queries use a k no more than 100. On the other hand, $k_{\max}$ does not seem to have a big impact on the construction time and the query performance (Figure 16(b) and Figure 16(c)). Hence, the SEB-trees still maintain their superior performance against the R-tree in these two categories, both by almost an order of magnitude. W.r.t. the MVB-tree solution, the SEB-trees are always almost two orders of magnitude more efficient to construct for different $k_{\max}$ values, and have achieved the similar query performance.

We next study the query performance of the SEB-tree when k becomes larger or even significantly larger than the $k_{\max}$ value used in its construction. Figure 16(d) shows the results where we test the query cost by varying the k value to as large as $6 \times k_{\max}$ ($k_{\max} = 200$ in this case), when the query performance of the basic SEB-tree eventually catches the query cost the R-tree for the first time.

We observed that the basic SEB-tree is highly robust. Even if k is significantly larger than $k_{\max}$ (as large as $3 \times k_{\max}$), the SEB-tree still has an impressive query performance and outperforms the R-tree solution by approximately one order of magnitude. The reason is that the SEB-tree is a randomized index structure. Our theoretical analysis on the size of the conflict list for each trapezoid built from the sampled envelope is based on the asymptotic result for the expectation. The constant in the big-O notation in this analysis has achieved a comfortable "cushion" effect. So that even if k is considerably larger than $k_{\max}$, in most cases, a conflict list retrieved from the B+ tree in the SEB-tree still has more than k objects.

We did notice that the SEB-tree variants, namely SEB-tree3 and SEB-tree4, are less robust towards large k values that are way beyond $k_{\max}$. Obviously, SEB-tree3 will be the least robust one since it keeps less number of layers. Our new experiments indicate that when k reaches $2k_{\max}$, SEB-tree3 becomes as bad as the R-tree solution. SEB-tree4's query performance also degrades as k keeps increasing beyond $k_{\max}$, however, even when $k = 2k_{\max}$, SEB-tree4 still outperforms the R-tree solution by a considerable margin (Figure 16(d) is plotted in the log-scale on the y-axis).

Finally, when k keeps increasing, the query costs for all solutions will increase. However, the query cost of the SEB-tree increases at a faster pace when k goes beyond $3 \times k_{\max}$, and it eventually catches the query cost of the R-tree when $k = 6 \times k_{\max}$. However, a too large k value is not useful in practice for most ranking applications. And, if this is really the case, any solution will become inefficient (as just to read and write the

output will be expensive). Hence, by choosing a k_{max} value that is comfortably larger than any foreseeable, useful k value will make sure that $k \gg k_{max}$ rarely happens. Note that this has little effect on the SEB-tree's performance as our experiment (Figures 16(a), 16(b) and 16(c)) shows that the choice of k_{max} does not have a significant impact on the size, the construction cost and the query cost of the SEB-tree and its variants. Furthermore, the result in Figure 16(d) ensures that even if k becomes considerably larger than k_{max} the basic SEB-tree will still have good performance. And, there will not be much room for k becoming one or two orders of magnitude larger than k_{max} before it already exceeds the number of objects in the database.

5.4 Update cost

Since the MVB-tree solution does not support the update operations, our final experiments study the update cost of the SEB-trees and compare it to the R-tree. Since insertions and deletions have similar results, we choose to report the results on the insertions here. We randomly generated 100 objects, each with 500 line segments, namely a total of 50,000 line segments for insertion. The results in Figure 17 report the average cost per insertion when we vary the total number of *existing objects* in the database before the insertions.

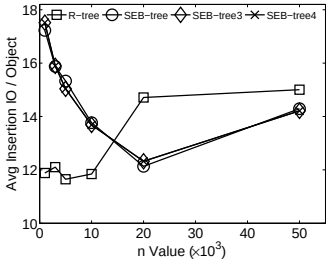


Fig. 17 Update I/O cost: vary n, $m = 500$, $\sigma = 30$, $N = n \times m$.

We observe an interesting results in Figure 17. First, inserting into an already larger R-tree is more costly, which is intuitive. However, the cost of inserting into the SEB-tree drops first and then starts to increase later. This counter-intuitive behavior can be explained by the following reasons. First of all, most of new segments to be inserted into the SEB-tree will not be in the sampled set. Hence, they will not update the envelope and the hierarchical trapezoidal decomposition. The chance that they do is only roughly $1/B$. Hence, the average insertion cost is dominated by these majority records. For each of these records, we simply need to insert it to the conflict lists of those trapezoids that it is conflicting with. That being said, the explanation to the trend in Figure 17 is similar to the reasons we have given for the index size behavior in Figure 9(a). As n gets larger, the trapezoids get smaller but we have more trapezoids. The former tends to reduce the number of conflicting trapezoids of a newly inserted segment, but the latter increases it. The former force is stronger for smaller n while the latter is stronger for larger n, which results in the interesting phenomenon we are observing in Figure 17. Nevertheless, the update cost of the R-tree and the SEB-tree is similar and both are less than 20 I/Os even for a database with $N = 25$ million line segments.

5.5 Summary of experimental results

These extensive experimental results have convincingly illustrated the superior performance of the SEB-tree for answering top-$k(t)$ queries, compared to the approaches that adopt existing indexing structures. In particular, the MVB-tree solution suffers from poor scalability for its index size and construction cost, and thus cannot be applied in real, large data sets. It also does not support general updates. The R-tree solution, on the other hand, suffers from the high query cost (by more than one order of magnitude) compared to the SEB-tree. In contrast, not only the SEB-tree achieves good construction cost, index size, query cost and update cost, but also it has a very simple structure. In fact, it is simply a collection of B-trees, which paves its way to practical deployments in existing DBMSs (no need to worry about locking and concurrency control issues that have been well addressed in commercial engines for B-trees).

6 Conclusion

This work studies ranking queries on temporal data. We introduce the SEB-tree that answers any top-$k(t)$ query with the optimal I/O cost in expectation. The SEB-tree takes near-linear time to construct and occupies near-linear space. It also supports dynamic updates efficiently. A nice feature of our approach is the fact that the SEB-tree employs the widely deployed B-tree as its only building block. This makes it not only easy to implement and deploy, but also extremely practical in practice, for supporting concurrency control, recovery and transactions. A number of interesting and important research directions are open, e.g., ranking queries within a temporal range rather than just at one time instance, ranking queries on original times series without segmentations, etc.

7 Acknowledgment

Feifei Li was partially supported by NSF Grant IIS-0916488. Ke Yi was supported in part by Hong Kong Direct Allocation Grant DAG07/08. Wangchao Le was supported in part by NSF CNS-0831278 grant.

References

1. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.
2. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.
3. P. K. Agarwal and M. Sharir. Davenport-Schinzel sequences and their geometric applications. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 1–47. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
4. C. C. Aggarwal and D. Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *PODS*, 2003.
5. A. Anagnostopoulos, M. Vlachos, M. Hadjieleftheriou, E. Keogh, and P. S. Yu. Global distance-based segmentation of trajectories. In *KDD*, 2006.
6. L. Arge, A. Danner, and S.-H. Teh. I/O-efficient point location using persistent B-trees. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2003.
7. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.
8. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.
9. Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *SIGMOD*, 2004.
10. T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. *SIAM Journal on Computing*, 30(2):561–575, 2000.
11. L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.
12. Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *VLDB*, 2007.
13. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Computational Geometry*, 4:387–421, 1989.
14. H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proc. VLDB Endow.*, 1(2):1542–1552, 2008.
15. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
16. E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Algorithms for nearest neighbor search on moving object trajectories. *Geoinformatica*, 11(2):159–193, 2007.
17. M. Hadjieleftheriou. The spatialindex library. www.research.att.com/∼marioh/spatialindex/index.html.
18. M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, 2005.
19. M. Hadjieleftheriou, G. Kollios, J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *The VLDB Journal*, 15(2):143–164, 2006.
20. S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6:151–177, 1986.
21. J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Inform. Process. Lett.*, 33:169–174, 1989.
22. I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):1–58, 2008.
23. C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B^+-tree based indexing of moving objects. In *VLDB*, 2004.
24. C. S. Jensen and D. B. Lomet. Transaction timestamping in (temporal) databases. In *VLDB*, 2001.
25. B. Jiang and J. Pei. Online interval skyline queries on time series. In *ICDE*, 2009.
26. E. Keogh, X. Xi, L. Wei, and C. Ratanamahatana. The UCR time series dataset, 2006. http://www.cs.ucr.edu/∼eamonn/time_series_data/.
27. E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An online algorithm for segmenting time series. In *ICDM*, 2001.
28. D. Lomet, R. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for sql server. In *SIGMOD*, 2005.
29. D. Lomet and F. Li. Improving transaction-time DBMS performance and functionality. In *ICDE*, 2009.
30. N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *KDD*, 2004.
31. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, 2004.
32. T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, 2004.
33. M. Pelanis, S. Šaltenis, and C. S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.*, 31(1):255–298, 2006.
34. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, 2000.
35. S. Ŝaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, 2000.
36. R. Sherkat and D. Rafiei. On efficiently searching trajectories and archival data for historical similarities. *Proc. VLDB Endow.*, 1(1):896–908, 2008.
37. J. Shieh and E. Keogh. iSAX: indexing and mining terabyte sized time series. In *KDD*, 2008.
38. Z. Song and N. Roussopoulos. SEB-tree: An approach to index continuously moving objects. In *MDM*, 2003.
39. Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, 2001.
40. Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, 2002.
41. B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, 2000.